Build your own Java library

Presented by developerWorks, your source for great tutorials

`ibm.com/developerWorks`

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Tutorial tips

Should I take this tutorial?

Code reuse is one of the Holy Grails of computer programming. Writing code that can be easily reused is a difficult skill, but certainly one that can be mastered.

In this tutorial, you will learn:

* How the Java language can help you make a good, reusable library
* A few of the key principles of good library design
* The most efficient way to implement each of these ideas in the Java language

To illustrate these ideas, we'll walk through the design of a simple library.

To use this tutorial successfully, you'll need to have a basic understanding of Java programming, including the ability to create, compile, and execute simple command-line Java programs.

Objectives

When you complete this tutorial, you will:

* Know the major obstacles to code reuse
* Have a working knowledge of the ideas behind library design
* Know how to implement these ideas in the Java language
* Possess an excellent design of a Java library as a sample for future projects

Getting help

For questions about the content of this tutorial, please contact the author, Greg Travis, at *mito@panix.com* .

Greg Travis is a freelance programmer living in New York City. His interest in computers can be traced back to that episode of "The Bionic Woman" where Jamie is trying to escape a building whose lights and doors are controlled by an evil artificial intelligence, which mocks her through loudspeakers. Greg is a firm believer that, when a computer program works, it's a complete coincidence.

Section 2. Introduction to libraries

What is a library?

A library is a reusable software component that saves developers time by providing access to the code that performs a programming task. Libraries exist to assist with many different types of tasks.

Library design is difficult. It's easy to take an algorithm you've written and call it a library, but it's harder to structure it in such a way that it can fit into someone else's program and still perform the tasks it is designed to do without interfering with the original program's operation.

Most modern languages try to help the programmer create good libraries, and the Java language is no exception. In this tutorial, you'll learn how the Java language can help you make a good, reusable library.

Why reuse code?

There is no sense in reinventing the wheel. If there is code that can be reused, it should be. Also, modern applications have become so large that it's inconceivable to even consider writing them without the use of libraries.

But that's not the only reason. Even if you have no intention of distributing a piece of code, creating it as a library can help you think more clearly about what it's supposed to do, thus helping refine the design process.

Why aren't libraries used more often?

The answer to this question is simple: if you don't write a library yourself, you have no control over the decisions that go into the crafting of it. And if the library behaves in a certain way that is not acceptable, you'll be less likely to use it. You could modify the library to suit your needs, but that might break compatibility. And, sometimes, rewriting something yourself is faster (and less stressful) than trying to understand someone else's code.

It is possible to make libraries simpler and more flexible, and therefore more useful for a wider range of programs. But no library is going to be perfect for every possible application. Still, it's possible to design and code libraries in such a way as to maximize their ease of use.

Put yourself in the user's shoes

When designing a library, it's important to put yourself in the place of the person who is going to be using the library. Ask these questions:

* What problem will the library solve? (For example, what should the end result be?)
* What details do the users care about? (For example, how do they want to get to the end results?)
* What details would the users rather forget about? (For example, what elements and

operations do they want other parts of the program to handle?)

In the final analysis, good library design involves a bit of psychology. Sure, your code works well, but does it *seem* like it works well? When people hear or read about it, will they say to themselves, "Wow, that's exactly what I need and exactly how I would have done it myself!"

A sample library

Our objective is to learn how the Java language can help to build an effective library. To achieve this goal, we'll discuss the design of a simple library that facilitates implementing network servers.

When creating a network server, there are a number of issues to consider:

* Listening on a socket
* Accepting connections
* Getting access to the streams represented by a connection
* Processing the incoming data in some way, and sending back a response

Our sample library is going to take care of the first three issues, leaving the fourth consideration for the library user to implement.

The main class in the library is called `Server`, and the test class that uses it is called `EchoServer`. The latter implements a simple, serial-connection server that takes data from the client and sends it right back. We'll talk about EchoServer on page 11in Section 6.

Design issues and practical implementations

The following sections cover the design issues of encapsulation, extensibility, and debugging. Each section deals with a single design issue and then discusses how the issue can be handled using the Java language.

The design principles themselves are not language dependent -- they can be applied to the design of libraries in any language.

Nevertheless, pay special attention to the implementation details; sometimes, they can illustrate a theoretical point better than a conceptual description can.

Section 3. Design issue: Encapsulation

What is encapsulation?

A library should function as a tight, self-contained unit rather than a scattered collection of objects whose functions and relationships are unclear.

The practice of making a library self-contained is called *encapsulation*.

What is a package?

The Java language provides an explicit mechanism for class-file-level encapsulation: *packages*. A package is a group of Java class files that are stored in a single directory; a package has a namespace to itself.

The advantage of giving a set of classes its own namespace is that you don't have to worry about namespace conflicts.

In our example, the main class has a fairly common name -- `Server`. It wouldn't be surprising if we eventually ran into a class in another library that had the same name. Putting the class in its own namespace takes care of any conflicts this might cause.

In the next few panels, I'll illustrate how to put a class into a package.

Packages have names

Every package has a name that consists of a set of strings separated by periods, such as `java.lang` or `javax.swing.plaf.basic`.

In fact, the full name of any class consists of the name of its package, followed by its own name, as in `java.lang.Object` or `javax.swing.plaf.basic.BasicMenuBarUI`.

Note that there is a special package called the *default package*. If you don't put your class into a specific package, then it is assumed to be in the default package.

Package names correspond to directories

Each package maps directly onto a subdirectory in the filesystem. This correspondence allows the Java virtual machine (JVM) to find classes at run time.

You can convert a package name into a subdirectory path by replacing the periods with "/" (or whatever symbol your operating system uses to separate directory names). For example, `java.lang.Object` is stored in the file `java/lang/Object.java`.

A class in the default package is placed in the current directory.

Classes declare their packages

To make sure a class is in the right package and in the right directory, a class must declare he package that it is in.

The declaration looks like this:

```
// Server.java

package mylib;

public class Server implements Runnable
{
  // ...
```

Other classes can import packages

To use a class that lives in another package, you can reference it by its full name, such as `mylib.Server`. For example:

```
  mylib.Server server = new mylib.Server( portNum );
```

It can be tedious to type out the full package name for a class, so you can take a shortcut and *import* the package like this:

```
import mylib.*;

  // ...

  Server server = new Server( portNum );
```

It's also possible to import a single class:

```
import mylib.Server;

  // ...

  Server server = new Server( portNum );
```

Choosing public classes

The Java language also allows you to decide which classes in a package are visible to the outside.

A *public* class can be accessed by code in any other package, while a *private* class can only be used within its own package.

It's important to make public *only those classes you want people to use directly*.

Designing the interface to a public class takes more care than designing one for a private class because the interface for a public class must be crafted to be as clear as possible for the user. A private class doesn't need to have as tidy an interface.

Making a class public requires using the `public` keyword on the first line of the class declaration:

```
// Server.java

package mylib;

import java.io.*;
import java.net.*;

public class Server implements Runnable
{
```

To make a class private, you simply leave the `public` keyword off. In our `Server` example, there's a private class in the same package called `Reporter`, which reports periodically on the condition of the `Server` object. Here is how it is declared:

```
// Reporter.java

package mylib;

class Reporter implements Runnable
{
```

Encapsulation summary

As we've seen, the Java language provides a few features that were created specifically for defining the boundaries of your code.

By dividing code into packages and by defining classes as public or private, you can make precise decisions about what the user of the library must deal with when they are using your library.

Section 4. Design issue: Extensibility

Inheritance

Encapsulation defines boundaries around a piece of code. All object-oriented programming languages provide a mechanism for extending a piece of code without violating these boundaries. In the Java language, this mechanism is provided by *inheritance*.

Customization through inheritance

The main class in our example library is called `Server`. If you look at the source code for this class, you'll see that, by itself, it does nothing.

The main loop (which runs in a separate thread) listens for incoming connections. When one comes in, it hands it off to a method called `handleConnection()`, as shown here:

```
// subclass must supply an implementation
abstract public void handleConnection( Socket s );
```

There is no default implementation, and so the class is declared `abstract`, which signals the user to supply an implementation. `EchoServer` implements this method:

```
// This is called by the Server class when a connection
// comes in.  "in" and "out" come from the incoming socket
// connection
public void handleConnection( Socket socket ) {
  try {
    InputStream in = socket.getInputStream();
    OutputStream out = socket.getOutputStream();

    // just copy the input to the output
    while (true)
      out.write( in.read() );

  } catch( IOException ie ) {
    System.out.println( ie );
  }
}
```

You can think of this process as a type of *customization*: the class that does most of the work (`Server`) is incomplete; the subclass (`EchoServer`) completes it by adding an implementation of the method `handleConnection()`.

Plan for hooks

It's important to think ahead to the kinds of customizations a user of your library is going to want to make. The previous example was obvious -- you have to provide a way for the subclass to actually use the incoming connections or your program won't do anything useful.

But if you don't think ahead, some other customizations might not occur to you. Instead, they'll occur to the user, who will wish that you had thought of them in the first place.

In our sample library, there is a method in `Server` called `cleanUp()`. This method is called by the server's background thread just before it exits. The base server doesn't need to do anything in this method, so it is empty:

```
// Put any last-minute clean-up stuff in here
protected void cleanUp() {
}
```

Note: Do not declare this method `abstract`, as it would require the user to implement this method, which, in some cases, isn't necessary.

Section 5. Design issue: Debugging information

Why should you plan for debugging?

A scenario: Your library works perfectly. The user needs to do nothing more than learn the API and use the code. Right?

Wrong. This scenario never (or rarely) happens. Bugs are unavoidable; debugging is inevitable.

At some point, a user is going to have a problem and will need to know exactly what is going on inside the library. The bug could be in the library itself, or it could be a bug in the user's code that is only *triggered* inside the library.

A little text goes a long way

If you supply the source with your library, a user *might* consider using a debugger, but you shouldn't count on that.

A better way to counter this inevitable problem is to add debugging `println()` statements to your code to force each piece of the code to report on what it is doing. Watching this information can often help the user know where something is going wrong.

The following example demonstrates this technique. The user's code can install a `PrintStream` object by calling the static method `Server.setDebugStream()`. Once implemented, the debug information will be sent to the provided stream.

```
// set this to a print stream if you want debug info
// sent to it; otherwise, leave it null
static private PrintStream debugStream;

// call this to send the debugging output somewhere
static public void setDebugStream( PrintStream ps ) {
  debugStream = ps;
}
```

You can then sprinkle your library with calls to the `debug()` method:

```
// send debug info to the print stream, if there is one
static public void debug( String s ) {
  if (debugStream != null)
    debugStream.println( s );
}
```

Section 6. The source code

Our sample library

Before we wrap up, take a look at the complete source code, which includes `EchoServer`, `mylib.Server`, and `mylib.Reporter`.

EchoServer

```
// $Id$

import java.io.*;
import java.net.*;
import mylib.*;

public class EchoServer extends Server
{
  public EchoServer( int port ) {
    // The superclass knows what to do with the port number, we
    // don't have to care about it
    super( port );
  }

  // This is called by the Server class when a connection
  // comes in.  "in" and "out" come from the incoming socket
  // connection
  public void handleConnection( Socket socket ) {
    try {
      InputStream in = socket.getInputStream();
      OutputStream out = socket.getOutputStream();

      // just copy the input to the output
      while (true)
        out.write( in.read() );

    } catch( IOException ie ) {
      System.out.println( ie );
    }
  }

  protected void cleanUp() {
    System.out.println( "Cleaning up" );
  }

  static public void main( String args[] ) throws Exception {
    // Grab the port number from the command-line
    int port = Integer.parseInt( args[0] );

    // Have debugging info sent to standard error stream
    Server.setDebugStream( System.err );

    // Create the server, and it's up and running
    new EchoServer( port );
  }
}
```

---

mylib.Server

```
// $Id$

package mylib;

import java.io.*;
import java.net.*;

abstract public class Server implements Runnable
{
  // the port we'll be listening on
  private int port;

  // how many connections we've handled
  int numConnections;

  // the Reporter that's reporting on this Server
  private Reporter reporter;

  // set this to true to tell the thread to stop accepting
  // connections
  private boolean mustQuit = false;

  public Server( int port ) {
    // remember the port number so the thread can
    // listen on it
    this.port = port;

    // the constructor starts a background thread
    new Thread( this ).start();

    // and start a reporter
    reporter = new Reporter( this );
  }

  // this is our background thread
  public void run() {
    ServerSocket ss = null;
    try {

      // get ready to listen
      ss = new ServerSocket( port );

      while( !mustQuit ) {

        // give out some debugging info
        debug( "Listening on "+port );

        // wait for an incoming connection
        Socket s = ss.accept();

        // record that we got another connection
        numConnections++;

        // more debugging info
        debug( "Got connection on "+s );

        // process the connection -- this is implemented
        // by the subclass
        handleConnection( s );
      }
    } catch( IOException ie ) {
      debug( ie.toString() );
    }

    debug( "Shutting down "+ss );

    cleanUp();
  }

  // the default implementation does nothing
  abstract public void handleConnection( Socket s );

  // tell the thread to stop accepting connections
  public void close() {
    mustQuit = true;
    reporter.close();
  }

  // Put any last-minute clean-up stuff in here
  protected void cleanUp() {
  }

  // everything below provides a simple debug system for
  // this package

  // set this to a print stream if you want debug info
  // sent to it; otherwise, leave it null
  static private PrintStream debugStream;

  // we have two versions of this ...
  static public void setDebugStream( PrintStream ps ) {
    debugStream = ps;
  }

  // ... just for convenience
  static public void setDebugStream( OutputStream out ) {
    debugStream = new PrintStream( out );
  }

  // send debug info to the print stream, if there is one
  static public void debug( String s ) {
    if (debugStream != null)
      debugStream.println( s );
  }
}
```

mylib.Reporter

```
// $Id$

package mylib;

class Reporter implements Runnable
{
  // the Server we are reporting on
  private Server server;

  // our background thread
  private Thread thread;

  // set this to true to tell the thread to stop accepting
  // connections
  private boolean mustQuit = false;

  Reporter( Server server ) {
    this.server = server;

   // create a background thread
    thread = new Thread( this );
    thread.start();
  }

  public void run() {
    while (!mustQuit) {
      // do the reporting
      Server.debug( "server has had "+server.numConnections+" connections" );

      // then pause a while
      try {
        Thread.sleep( 5000 );
      } catch( InterruptedException ie ) {}
    }
  }

  // tell the background thread to quit
  public void close() {
    mustQuit = true;
  }
}
```

Section 7. Wrapup

Revisiting the design issues

In this tutorial, we've looked at some concrete ways in which the Java language eases the creation of a good, reusable library. We've also explored some of the theory upon which these language features are based.

A good library is one that conveys its internal structure clearly so that the user knows how to use it without knowing how it works internally.

Encapsulation serves the purpose of tightening and clarifying the interface to your code, which makes it more reliable and easier to understand.

Extensibility allows you to create a library that does a single thing well, leaving the user to fill in the missing parts and to re-purpose the library to meet user-specific needs. Because the Java language is object oriented, the method is clear: inheritance provides a way to *customize* code without having to know all the details of the code you are customizing.

Finally, providing a way to get **debugging information** from your code helps users find their own bugs (or yours!), again without having to know every detail of the library's implementation.

Resources

Following is a listing of additional readings to help solidify your understanding of libraries:

* Take a look at " *Techniques for adding trace statements to your Java application* " (developerWorks, January 2001) for some more advanced ideas about adding extensive debugging output to your library.
* Peter Haggar offers advice for *how to use the Java libraries judiciously* (IBM PartnerWorld for Developers).
* Frameworks are an excellent conceptual tool for designing reusable code. " *Frameworks save the day* " (developerWorks, October 2000) explores frameworks using Java technology.
* *The October 1998 issue* of *Web Techniques* is dedicated to Java frameworks.
* Brush up on `import` and `packages` in the *The Java Language Specification* .

Your feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered. Thanks!

For questions about the content of this tutorial, contact the author, Greg Travis, at *mito@panix.com* .

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.